

# A faster exact multiprocessor schedulability test for sporadic tasks

Markus Lindström

Gilles Geeraerts

Joël Goossens

Université libre de Bruxelles  
Département d'Informatique, Faculté des Sciences  
Avenue Franklin D. Roosevelt 50, CP 212  
1050 Bruxelles, Belgium  
{mlindstr, gilles.geeraerts, joel.goossens}@ulb.ac.be

## Abstract

*Baker and Cirinei introduced an exact but naive algorithm [3], based on solving a state reachability problem in a finite automaton, to check whether sets of sporadic hard real-time tasks are schedulable on identical multiprocessor platforms. However, the algorithm suffered from poor performance due to the exponential size of the automaton relative to the size of the task set. In this paper, we successfully apply techniques developed by the formal verification community, specifically antichain algorithms [11], by defining and proving the correctness of a simulation relation on Baker and Cirinei's automaton. We show our improved algorithm yields dramatically improved performance for the schedulability test and opens for many further improvements.*

## 1. Introduction

In this research we consider the schedulability problem of hard real-time sporadic constrained deadline task systems upon identical multiprocessor platforms. Hard real-time systems are systems where tasks are not only required to provide correct computations but are also required to adhere to strict deadlines [16].

Devising an exact schedulability criterion for sporadic task sets on multiprocessor platforms has so far proven difficult due to the fact that there is no known worst case scenario (nor critical instant). It was notably shown in [14] that the periodic case is not necessarily the worst on multiprocessor systems. In this context, the real-time community has mainly been focused on the development of *sufficient* schedulability tests that correctly identify all unschedulable task sets, but may misidentify some schedulable systems as being unschedulable [2] using a given platform and scheduling policy (see e.g. [5, 4]).

Baker and Cirinei introduced the first correct algorithm [3] that verified *exactly* whether a sporadic task system was schedulable on an identical multiprocessor platform by solving a reachability problem on a finite state

automaton using a naive brute-force algorithm, but it suffered from the fact that the number of states was exponential in the size of the task sets and its periods, which made the algorithm intractable even for small task sets with large enough periods.

In this paper, we apply techniques developed by the formal verification community, specifically Doyen, Raskin *et al.* [11, 9] who developed faster algorithms to solve the reachability problem using algorithms based on data structures known as *antichains*. Their method has been shown to be provably better [11] than naive state traversal algorithms such as those used in [3] for deciding reachability from a set of initial states to a given set of final states.

An objective of this work is to be as self-contained as possible to allow readers from the real-time community to be able to fully understand the concepts borrowed from the formal verification community. We also hope our work will kickstart a “specialisation” of the methods presented herein within the realm of real-time scheduling, thus bridging the two communities.

**Related work.** This work is not the first contribution to apply techniques and models first proposed in the setting of formal verification to real-time scheduling. In the field of operational research, Abdeddaïm and Maler have studied the use of stopwatch automata to solve job-shop scheduling problems [1]. Cassez has recently exploited game theory, specifically timed games, to bound worst-case execution times on modern computer architectures, taking into account caching and pipelining [8]. Fersman *et al.* have studied a similar problem and introduced task automata which assume continuous time [12], whereas we consider discrete time in our work. They showed that, given selected constraints, schedulability could be undecidable in their model. Bonifaci and Marchetti-Spaccamela have studied the related problem of feasibility of multiprocessor sporadic systems in [6] and have established an upper bound on its complexity.

**This research.** We define a restriction to constrained deadlines (systems where the relative deadline of tasks is no longer than their minimal interarrival time) of Baker and Cirinei’s automaton in a more formal way than in [3]. We also formulate various scheduling policy properties in the framework of this automaton such as memorylessness.

Our main contribution is the design and proof of correctness of a non-trivial *simulation relation* on the automaton, required to successfully apply a generic algorithm developed in the formal verification community, known as an *antichain algorithm* to Baker and Cirinei’s automaton to prove or disprove the schedulability of a given sporadic task system.

Finally, we will show through implementation and experimental analysis that our proposed algorithm outperforms Baker and Cirinei’s original brute-force algorithm.

**Paper organization.** Section 2 defines the real-time scheduling problem we are focusing on, i.e. devising an exact schedulability test for sporadic task sets on identical multiprocessor platforms. Section 3 will formalize the model (a non-deterministic automaton) we will use to describe the problem and we formulate how the schedulability test can be mapped to a reachability problem in this model. We also formalize various real-time scheduling concepts in the framework of our formal model.

Section 4 then discusses how the reachability problem can be solved. We present the classical breadth-first algorithm used in [3] and we introduce an improved algorithm that makes use of techniques borrowed from the formal verification community [11]. The algorithm requires coarse *simulation relations* to work faster than the standard breadth-first algorithm. Section 5 introduces the *idle tasks simulation relation* which can be exploited by the aforementioned algorithm.

Section 6 then showcases experimental results comparing the breadth-first and our improved algorithm using the aforementioned simulation relation, showing that our algorithm outperforms the naive one. Section 7 concludes our work. Appendix A gives a detailed proof of a lemma we use in Section 4.

## 2. Problem definition

We consider an identical multiprocessor platform with  $m$  processors and a sporadic task set  $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$ . Time is assumed to be discrete. A sporadic task  $\tau_i$  is characterized by a *minimum interarrival time*  $T_i > 0$ , a *relative deadline*  $D_i > 0$  and a *worst-case execution time* (also written WCET)  $C_i > 0$ . A sporadic task  $\tau_i$  submits a potentially infinite number of jobs to the system, with each request being separated by at least  $T_i$  units of time. We will assume jobs are not parallel, i.e. only execute on one single processor (though it may migrate from a processor to another during execution). We also assume jobs are independent. We wish to establish an *exact* schedulability test for any sporadic task set  $\tau$  that tells us whether the set

is schedulable on the platform with a given deterministic, predictable and preemptive scheduling policy. In the remainder of this paper, we will assume we only work with *constrained deadline* systems (i.e. where  $\forall \tau_i : D_i \leq T_i$ ) which embody many real-time systems in practice.

## 3. Formal definition of the Baker-Cirinei automaton

Baker and Cirinei’s automaton as presented in [3] models the evolution of an *arbitrary* deadline sporadic task set (with a FIFO policy for jobs of a given task) scheduled on an identical multiprocessor platform with  $m$  processors. In this paper, we focus on constrained deadline systems as this hypothesis simplifies the definition of the automaton. We expect to analyze Baker and Cirinei’s more complete construct in future works.

The model presented herein allows use of *preemptive*, *deterministic* and *predictable* scheduling policies. It can, however, be generalized to model broader classes of schedulers. We will discuss this aspect briefly in Section 7.

**Definition 1.** An *automaton* is a tuple  $A = \langle V, E, S_0, F \rangle$ , where  $V$  is a finite set of *states*,  $E \subseteq V \times V$  is the set of *transitions*,  $S_0 \in V$  is the *initial state* and  $F \subseteq V$  is a set of *target states*.

The problem on automata we are concerned with is that of *reachability* (of target states). A *path* in an automaton  $A = \langle V, E, S_0, F \rangle$  is a finite sequence  $v_1, \dots, v_\ell$  of states s.t. for all  $1 \leq i \leq \ell - 1$ :  $(v_i, v_{i+1}) \in E$ . Let  $V' \subseteq V$  be a set of states of  $A$ . If there exists a path  $v_1, \dots, v_\ell$  in  $A$  s.t.  $v_\ell \in V'$ , we say that  $v_1$  *can reach*  $V'$ . Then, the *reachability problem* asks, given an automaton  $A$  whether the initial state  $S_0$  can reach the set of target states  $F$ .

Let  $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$  be a set of sporadic tasks and  $m$  be a number of processors. This section is devoted to explaining how to model the behaviour of such a system by means of an automaton  $A$ , and how to reduce the schedulability problem of  $\tau$  on  $m$  processors to an instance of the reachability problem in  $A$ . At any moment during the execution of such a system, the information we need to retain about each task  $\tau_i$  are: (i) the *earliest next arrival time*  $\text{nat}(\tau_i)$  relative to the current instant and (ii) the remaining processing time  $\text{rct}(\tau_i)$  of the currently ready job of  $\tau_i$ . Hence the definition of *system state*:

**Definition 2** (System states). Let  $\tau = \{\tau_1, \tau_2, \dots, \tau_n\}$  be a set of sporadic tasks. A *system state* of  $\tau$  is a tuple  $S = \langle \text{nat}_S, \text{rct}_S \rangle$  where  $\text{nat}_S$  is a function from  $\tau$  to  $\{0, 1, \dots, T_{\max}\}$  where  $T_{\max} \stackrel{\text{def}}{=} \max_i T_i$ , and  $\text{rct}_S$  is a function from  $\tau$  to  $\{0, 1, \dots, C_{\max}\}$ , where  $C_{\max} \stackrel{\text{def}}{=} \max_i C_i$ . We denote by  $\text{States}(\tau)$  the set of all system states of  $\tau$ .

In order to define the set of transitions of the automaton, we need to rely on ancillary notions:

**Definition 3** (Eligible task). A task  $\tau_i$  is *eligible* in the state  $S$  if it can submit a job (i.e. if and only if the task does not currently have an active job and the last job was submitted at least  $T_i$  time units ago) from this configuration. Formally, the set of eligible tasks in state  $S$  is:

$$\text{Eligible}(S) \stackrel{\text{def}}{=} \{\tau_i \mid \text{nat}_S(\tau_i) = \text{rct}_S(\tau_i) = 0\}$$

**Definition 4** (Active task). A task is *active* in state  $S$  if it currently has a job that has not finished in  $S$ . Formally, the set of active tasks in  $S$  is:

$$\text{Active}(S) \stackrel{\text{def}}{=} \{\tau_i \mid \text{rct}_S(\tau_i) > 0\}$$

A task that is not active in  $S$  is said to be *idle* in  $S$ .

**Definition 5** (Laxity [3]). The laxity of a task  $\tau_i$  in a system state  $S$  is:

$$\text{laxity}_S(\tau_i) \stackrel{\text{def}}{=} \text{nat}_S(\tau_i) - (T_i - D_i) - \text{rct}_S(\tau_i)$$

**Definition 6** (Failure state). A state  $S$  is a *failure state* iff the laxity of at least one task is negative in  $S$ . Formally, the set of failure states on  $\tau$  is:

$$\text{Fail}_\tau \stackrel{\text{def}}{=} \{S \mid \exists \tau_i \in \tau : \text{laxity}_S(\tau_i) < 0\}$$

Thanks to these notions we are now ready to explain how to build the transition relation of the automaton that models the behaviour of  $\tau$ . For that purpose, we first choose a *scheduler*. Intuitively, a scheduler is a *function*<sup>1</sup>  $\text{Run}$  that maps each state  $S$  to a set of at most  $m$  active tasks  $\text{Run}(S)$  to be run:

**Definition 7** (Scheduler). A (deterministic) *scheduler* for  $\tau$  on  $m$  processors is a function  $\text{Run} : \text{States}(\tau) \rightarrow 2^\tau$  s.t. for all  $S$ :  $\text{Run}(S) \subseteq \text{Active}(S)$  and  $0 \leq |\text{Run}(S)| \leq m$ . Moreover:

1.  $\text{Run}$  is *work-conserving* iff for all  $S$ ,  $|\text{Run}(S)| = \min\{m, |\text{Active}(S)|\}$
2.  $\text{Run}$  is *memoryless* iff for all  $S_1, S_2 \in \text{States}(\tau)$  with  $\text{Active}(S_1) = \text{Active}(S_2)$ :

$$\forall \tau_i \in \text{Active}(S_1) : \left( \begin{array}{l} \text{nat}_{S_1}(\tau_i) = \text{nat}_{S_2}(\tau_i) \\ \wedge \text{rct}_{S_1}(\tau_i) = \text{rct}_{S_2}(\tau_i) \end{array} \right) \text{ implies } \text{Run}(S_1) = \text{Run}(S_2)$$

Intuitively, the work-conserving property implies that the scheduler always exploits as many processors as available. The memoryless property implies that the decisions of the scheduler are not affected by tasks that are idle and that the scheduler does not consider the past to make its decisions.

As examples, we can formally define the preemptive global DM and EDF schedulers.

<sup>1</sup>Remark that by modeling the scheduler as a function, we restrict ourselves to *deterministic schedulers*.

**Definition 8** (Preemptive global DM scheduler). Let  $\ell \stackrel{\text{def}}{=} \min\{m, |\text{Active}(S)|\}$ . Then,  $\text{Run}_{\text{DM}}$  is a function that computes  $\text{Run}_{\text{DM}}(S) \stackrel{\text{def}}{=} \{\tau_{i_1}, \tau_{i_2}, \dots, \tau_{i_\ell}\}$  s.t. for all  $1 \leq j \leq \ell$  and for all  $\tau_k$  in  $\text{Active}(S) \setminus \text{Run}_{\text{DM}}(S)$ , we have  $D_k > D_{i_j}$  or  $D_k = D_{i_j} \wedge k > i_j$ .

**Definition 9** (Preemptive global EDF scheduler). Let  $\text{ttd}_S(\tau_i) \stackrel{\text{def}}{=} \text{nat}_S(\tau_i) - (T_i - D_i)$  be the time remaining before the absolute deadline of the last submitted job [3] of  $\tau_i \in \text{Active}(S)$  in state  $S$ . Let  $\ell \stackrel{\text{def}}{=} \min\{m, |\text{Active}(S)|\}$ . Then,  $\text{Run}_{\text{EDF}}$  is a function that computes  $\text{Run}_{\text{EDF}}(S) \stackrel{\text{def}}{=} \{\tau_{i_1}, \tau_{i_2}, \dots, \tau_{i_\ell}\}$  s.t. for all  $1 \leq j \leq \ell$  and for all  $\tau_k$  in  $\text{Active}(S) \setminus \text{Run}_{\text{EDF}}(S)$ , we have  $\text{ttd}_S(\tau_k) > \text{ttd}_S(\tau_{i_j})$  or  $\text{ttd}_S(\tau_k) = \text{ttd}_S(\tau_{i_j}) \wedge k > i_j$ .

By Definition 7, global DM and EDF are thus work-conserving and it can also be verified that they are memoryless. In [3], suggestions to model several other schedulers were presented. It was particularly shown that adding supplementary information to system states could allow broader classes of schedulers to be used. Intuitively, states could e.g. keep track of what tasks were executed in their predecessor to implement non-preemptive schedulers.

Clearly, in the case of the scheduling of sporadic tasks, two types of events can modify the current state of the system:

1. *Clock-tick transitions* model the elapsing of time for one time unit, i.e. the execution of the scheduler and the running of jobs.
2. *Request transitions* (called *ready transitions* in [3]) model requests from sporadic tasks at a given instant in time.

Let  $S$  be a state in  $\text{States}(\tau)$ , and let  $\text{Run}$  be a scheduler. Then, letting one time unit elapse from  $S$  under the scheduling policy imposed by  $\text{Run}$  amounts to decrementing the  $\text{rct}$  of the tasks in  $\text{Run}(S)$  (and only those tasks), and to decrementing the  $\text{nat}$  of all tasks. Formally:

**Definition 10.** Let  $S = \langle \text{nat}_S, \text{rct}_S \rangle \in \text{States}(\tau)$  be a system state and  $\text{Run}$  be a scheduler for  $\tau$  on  $m$  processors. Then, we say that  $S^+ = \langle \text{nat}_S^+, \text{rct}_S^+ \rangle$  is a *clock-tick successor* of  $S$  under  $\text{Run}$ , denoted  $S \xrightarrow{\text{Run}} S^+$  iff:

1. for all  $\tau_i \in \text{Run}(S)$ :  $\text{rct}_S^+(\tau_i) = \text{rct}_S(\tau_i) - 1$  ;
2. for all  $\tau_i \notin \text{Run}(S)$ :  $\text{rct}_S^+(\tau_i) = \text{rct}_S(\tau_i)$  ;
3. for all  $\tau_i \in \tau$ :  $\text{nat}_S^+(\tau_i) = \max\{\text{nat}_S(\tau_i) - 1, 0\}$ .

Let  $S$  be a state in  $\text{States}(\tau)$ . Intuitively, when the system is in state  $S$ , a request by some task  $\tau_i$  for submitting a new job has the effect to update  $S$  by setting  $\text{nat}(\tau_i)$  to  $T_i$  and  $\text{rct}(\tau_i)$  to  $C_i$ . This can be generalised to sets of tasks. Formally:

**Definition 11.** Let  $S \in \text{States}(\tau)$  be a system state and let  $\tau' \subseteq \text{Eligible}(S)$  be a set of tasks that are eligible to submit a new job in the system. Then, we say that  $S'$  is a  $\tau'$ -request successor of  $S$ , denoted  $S \xrightarrow{\tau'} S'$ , iff:

1. for all  $\tau_i \in \tau'$ :  $\text{nat}_{S'}(\tau_i) = T_i$  and  $\text{rct}_{S'}(\tau_i) = C_i$
2. for all  $\tau_i \in \tau \setminus \tau'$ :  $\text{nat}_{S'}(\tau_i) = \text{nat}_S(\tau_i)$  and  $\text{rct}_{S'}(\tau_i) = \text{rct}_S(\tau_i)$ .

Remark that we allow  $\tau' = \emptyset$  (that is, no task asks to submit a new job in the system).

We are now ready to define the automaton  $A(\tau, \text{Run})$  that formalises the behavior of the system of sporadic tasks  $\tau$ , when executed upon  $m$  processors under a scheduling policy  $\text{Run}$ :

**Definition 12.** Given a set of sporadic tasks  $\tau$  and a scheduler  $\text{Run}$  for  $\tau$  on  $m$  processors, the automaton  $A(\tau, \text{Run})$  is the tuple  $\langle V, E, S_0, F \rangle$  where:

1.  $V = \text{States}(\tau)$
2.  $(S_1, S_2) \in E$  iff there exists  $S' \in \text{States}(\tau)$  and  $\tau' \subseteq \tau$  s.t.  $S_1 \xrightarrow{\tau'} S' \xrightarrow{\text{Run}} S_2$ .
3.  $S_0 = \langle \text{nat}_0, \text{rct}_0 \rangle$  where for all  $\tau_i \in \tau$ ,  $\text{nat}_0(\tau_i) = \text{rct}_0(\tau_i) = 0$ .
4.  $F = \text{Fail}_\tau$

Figure 1 illustrates a possible graphical representation of one such automaton, which will be analyzed further in Section 5. On this example, the automaton depicts the following EDF-schedulable sporadic task set using an EDF scheduler and assuming  $m = 2$ :

	$T_i$	$D_i$	$C_i$
$\tau_1$	2	2	1
$\tau_2$	3	3	2

System states are represented by nodes. For the purpose of saving space, we represent a state  $S$  with the  $[\alpha\beta, \gamma\delta]$  format, meaning  $\text{nat}_S(\tau_1) = \alpha$ ,  $\text{rct}_S(\tau_1) = \beta$ ,  $\text{nat}_S(\tau_2) = \gamma$  and  $\text{rct}_S(\tau_2) = \delta$ . We explicitly represent clock-tick transitions by edges labelled with  $\text{Run}$ , and  $\tau'$ -request transitions by edges labelled with  $\tau'$ .  $\tau' = \emptyset$  loops are implicit on each state. Note that, in accordance with Definition 12, there are no successive  $\tau'$ -request transitions, and there are thus no such transitions from states such as  $[21, 00]$  and  $[00, 32]$ . Also note that the automaton indeed models the evolution of a sporadic system, of which the periodic case is one possible path (the particular case of a synchronous system is found by taking the maximal  $\tau'$ -request transition whenever possible, starting from  $[00, 00]$ ).

We remark that our definition deviates slightly from that of Baker and Cirinei. In our definition, a path in the automaton corresponds to an execution of the system that alternates between requests transitions (possibly with an empty set of requests) and clock-tick transitions. In their

work [3], Baker and Cirinei allow any sequence of clock ticks and requests, but restrict each request to a single task at a time. It is easy to see that these two definitions are equivalent. A sequence of  $k$  clock ticks in Baker's automaton corresponds in our case to a path  $S_1, S_2, \dots, S_{k+1}$  s.t. for all  $1 \leq i \leq k$ :  $S_i \xrightarrow{\emptyset} S_i \xrightarrow{\text{Run}} S_{i+1}$ . A maximal sequence of successive requests by  $\tau_1, \tau_2, \dots, \tau_k$ , followed by a clock tick corresponds in our case to a single edge  $(S_1, S_2)$  s.t.  $S_1 \xrightarrow{\{\tau_1, \dots, \tau_k\}} S' \xrightarrow{\text{Run}} S_2$  for some  $S'$ . Conversely, each edge  $(S_1, S_2)$  in  $A(\tau, \text{Run})$  s.t.  $S_1 \xrightarrow{\tau'} S' \xrightarrow{\text{Run}} S_2$ , for some state  $S'$  and set of tasks  $\tau' = \{\tau_1, \dots, \tau_k\}$ , corresponds to a sequence of successive requests<sup>2</sup> by  $\tau_1, \dots, \tau_k$  followed by a clock tick in Baker's setting.

The purpose of the definition of  $A(\tau, \text{Run})$  should now be clear to the reader. Each possible execution of the system corresponds to a path in  $A(\tau, \text{Run})$  and vice-versa. States in  $\text{Fail}_\tau$  correspond to states of the system where a deadline will unavoidably be missed. Hence, *the set of sporadic tasks  $\tau$  is feasible under scheduler  $\text{Run}$  on  $m$  processors iff  $\text{Fail}_\tau$  is not reachable in  $A(\tau, \text{Run})$*  [3]. Unfortunately, the number of states of  $A(\tau, \text{Run})$  can be intractable even for very small sets of tasks  $\tau$ . In the next section we present generic techniques to solve the reachability problem in an efficient fashion, and apply them to our case. Experimental results given in Section 6 demonstrate the practical interest of these methods.

## 4. Solving the reachability problem

Let us now discuss techniques to solve the reachability problem. Let  $A = \langle V, E, S_0, F \rangle$  be an automaton. For any  $S \in V$ , let  $\text{Succ}(S) = \{S' \mid (S, S') \in E\}$  be the set of one-step successors of  $S$ . For a set of states  $R$ , we let  $\text{Succ}(R) = \cup_{S \in R} \text{Succ}(S)$ . Then, solving the reachability problem on  $A$  can be done by a *breadth-first traversal* of the automaton, as shown in Algorithm 1.

---

### Algorithm 1: Breadth-first traversal.

---

```

1 begin
2    $i \leftarrow 0$ ;
3    $R_0 \leftarrow \{S_0\}$ ;
4   repeat
5      $i \leftarrow i + 1$ ;
6      $R_i \leftarrow R_{i-1} \cup \text{Succ}(R_{i-1})$ ;
7     if  $R_i \cap F \neq \emptyset$  then return Reachable;
8   until  $R_i = R_{i-1}$ ;
9   return Not reachable;

```

---

Intuitively, for all  $i \geq 0$ ,  $R_i$  is the set of states that are reachable from  $S_0$  in  $i$  steps at most. The algorithm computes the sets  $R_i$  up to the point where (i) either a state from  $F$  is met or (ii) the sequence of  $R_i$  stabilises because no new states have been discovered, and we declare

<sup>2</sup>Remark that the order does not matter.

$F$  to be unreachable. This algorithm always terminates and returns the correct answer. Indeed, either  $F$  is reachable in, say  $k$  steps, and then  $R_k \cap F \neq \emptyset$ , and we return ‘Reachable’. Or  $F$  is not reachable, and the sequence eventually stabilises because  $R_0 \subseteq R_1 \subseteq R_2 \subseteq \dots \subseteq V$ , and  $V$  is a finite set. Then, we exit the loop and return ‘Not reachable’. Remark that this algorithm has the advantage that the whole automaton does not need be stored in memory before starting the computation, as Definition 10 and Definition 11 allow us to compute  $\text{Succ}(S)$  *on the fly* for any state  $S$ . Nevertheless, in the worst case, this procedure needs to explore the whole automaton and is thus in  $\mathcal{O}(|V|)$  which can be too large to handle in practice [3].

Equipped with such a simple definition of *automaton*, this is the best algorithm we can hope for. However, in many practical cases, the set of states of the automaton is endowed with a *strong semantic* that can be exploited to speed up Algorithm 1. In our case, states are tuples of integers that characterise sporadic tasks running in a system. To harness this information, we rely on the formal notion of *simulation*:

**Definition 13.** Let  $A = \langle V, E, S_0, F \rangle$  be an automaton. A *simulation relation* for  $A$  is a preorder  $\succsim \subseteq V \times V$  s.t.:

1. For all  $S_1, S_2, S_3$  s.t.  $(S_1, S_2) \in E$  and  $S_3 \succsim S_1$ , there exists  $S_4$  s.t.  $(S_3, S_4) \in E$  and  $S_4 \succsim S_2$ .
2. For all  $S_1, S_2$  s.t.  $S_1 \succsim S_2$ :  $S_2 \in F$  implies  $S_1 \in F$ .

Whenever  $S_1 \succsim S_2$ , we say that  $S_1$  *simulates*  $S_2$ . Whenever  $S_1 \succsim S_2$  but  $S_2 \not\succsim S_1$ , we write  $S_1 \succ S_2$ .

Intuitively, this definition says that whenever a state  $S_3$  simulates a state  $S_1$ , then  $S_3$  can *mimick* every possible move of  $S_1$  by moving to a similar state: for every edge  $(S_1, S_2)$ , there is a corresponding edge  $(S_3, S_4)$ , where  $S_4$  simulates  $S_2$ . Moreover, we request that a *target state* can only be simulated by a target state. Remark that for a given automaton there can be several simulation relations (for instance, equality is always a simulation relation).

The key consequence of this definition is that **if**  $S_2$  is a state that can reach  $F$ , and if  $S_1 \succ S_2$  **then**  $S_1$  can reach  $F$  too. Indeed, if  $S_2$  can reach  $F$ , there is a path  $v_0, v_1, \dots, v_n$  with  $v_0 = S_2$  and  $v_n \in F$ . Using Definition 13 we can inductively build a path  $v'_0, v'_1, \dots, v'_n$  s.t.  $v'_0 = S_1$  and  $v'_i \succ v_i$  for all  $i \geq 0$ . Thus, in particular  $v'_n \succ v_n \in F$ , hence  $v'_n \in F$  by Definition 13. This means that  $S_1$  can reach  $F$  too. Thus, when we compute two states  $S_1$  and  $S_2$  with  $S_1 \succ S_2$ , at some step of Algorithm 1, we *do not need to further explore the successors of*  $S_2$ . Indeed, Algorithm 1 tries to detect reachable target states. So, if  $S_2$  cannot reach a failure state, it is safe not to explore its successors. Otherwise, if  $S_2$  can reach a target state, then  $S_1$  can reach a target state too, so it is safe to explore the successors of  $S_1$  only. By exploiting this heuristic, Algorithm 1 could explore only a (small) subset of the states of  $A$ , which has the potential for a

dramatic improvement in computation time. Remark that such techniques have already been exploited in the setting of *formal verification*, where several so-called *antichains algorithms* have been studied [9, 11, 13] and have proved to be *several order of magnitudes more efficient* than the classical techniques of the literature.

Formally, for a set of states  $V' \subseteq V$ , we let  $\text{Max}^\succ(V') = \{S \in V' \mid \nexists S' \in V' \text{ with } S' \succ S\}$ . Intuitively,  $\text{Max}^\succ(V')$  is obtained from  $V'$  by removing all the states that are simulated by some other state in  $V'$ . So the states we keep in  $\text{Max}^\succ(V')$  are *irredundant*<sup>3</sup> wrt  $\succ$ . Then, we consider Algorithm 2 which is an improved version of Algorithm 1.

---

**Algorithm 2:** Improved breadth-first traversal.

---

```

1 begin
2    $i \leftarrow 0$ ;
3    $\tilde{R}_0 \leftarrow \{S_0\}$ ;
4   repeat
5      $i \leftarrow i + 1$ ;
6      $\tilde{R}_i \leftarrow \tilde{R}_{i-1} \cup \text{Succ}(\tilde{R}_{i-1})$ ;
7      $\tilde{R}_i \leftarrow \text{Max}^\succ(\tilde{R}_i)$ ;
8     if  $\tilde{R}_i \cap F \neq \emptyset$  then return Reachable;
9   until  $\tilde{R}_i = \tilde{R}_{i-1}$ ;
10  return Not reachable;

```

---

Proving the correctness and termination of Algorithm 2 is a little bit more involved than for Algorithm 1 and relies on the following lemma (proof in appendix):

**Lemma 14.** Let  $A$  be an automaton and let  $\succsim$  be a simulation relation for  $A$ . Let  $R_0, R_1, \dots$  and  $\tilde{R}_0, \tilde{R}_1, \dots$  denote respectively the sequence of sets computed by Algorithm 1 and Algorithm 2 on  $A$ . Then, for all  $i \geq 0$ :  $\tilde{R}_i = \text{Max}^\succ(R_i)$ .

Intuitively, this means that some state  $S$  that is in  $R_i$  could not be present in  $\tilde{R}_i$ , but that we always keep in  $\tilde{R}_i$  a state  $S'$  that simulates  $S$ . Then, we can prove that:

**Theorem 15.** For all automata  $A = \langle V, E, S_0, F \rangle$ , Algorithm 2 terminates and returns “Reachable” iff  $F$  is reachable in  $A$ .

*Proof.* The proof relies on the comparison between the sequence of sets  $R_0, R_1, \dots$  computed by Algorithm 1 (which is correct and terminates) and the sequence  $\tilde{R}_0, \tilde{R}_1, \dots$  computed by Algorithm 2.

Assume  $F$  is reachable in  $A$  in  $k$  steps and not reachable in less than  $k$  steps. Then, there exists a path  $v_0, v_1, \dots, v_k$  with  $v_0 = S_0, v_k \in F$ , and, for all  $0 \leq i \leq k$   $v_i \in R_k$ . Let us first show *per absurdum* that the loop in Algorithm 2 does not finish before the  $k$ th step. Assume it is not the case, i.e. there exists  $0 < \ell < k$  s.t.  $\tilde{R}_\ell = \tilde{R}_{\ell-1}$ .

---

<sup>3</sup>They form an *antichain* of states wrt  $\succ$ .

This implies that  $\text{Max}^{\succ}(R_\ell) = \text{Max}^{\succ}(R_{\ell-1})$  through Lemma 14. Since  $R_\ell \neq R_{\ell-1}$ , we deduce that all the states that have been added to  $R_\ell$  are simulated by some state already present in  $R_{\ell-1}$ : for all  $S \in R_\ell$ , there is  $S' \in R_{\ell-1}$  s.t.  $S' \succ S$ . Thus, in particular, there is  $S' \in R_{\ell-1}$  s.t.  $S' \succ v_\ell$ . We consider two cases. Either there is  $S' \in R_{\ell-1}$  s.t.  $S' \succ v_k$ . Since  $v_k \in F$ ,  $F \cap R_{\ell-1} \neq \emptyset$ , which contradicts our hypothesis that  $F$  is not reachable in less than  $k$  steps. Otherwise, let  $0 \leq m < k$  be the least position in the path s.t. there is  $S' \in R_{\ell-1}$  with  $S' \succ v_m$ , but there is no  $S'' \in R_{\ell-1}$  with  $S'' \succ v_{m+1}$ . In this case, since  $S' \succ v_m$  and  $(v_m, v_{m+1}) \in E$ , there is  $S \in \text{Succ}(S') \subseteq R_\ell$  s.t.  $S \succ v_{m+1}$ . However, we have made the hypothesis that every element in  $R_\ell$  is simulated by some element in  $R_{\ell-1}$ . Thus, there is  $S'' \in R_{\ell-1}$  s.t.  $S'' \succ S$ . Since  $S \succ v_{m+1}$ , we deduce that  $S'' \succ v_{m+1}$ , with  $S'' \in R_{\ell-1}$ , which contradicts our assumption that  $S'' \notin R_{\ell-1}$ . Thus, Algorithm 2 will not stop before the  $k$ th iteration, and we know that there is  $S_F \in R_k$  s.t.  $S_F \in F$ . By Lemma 14,  $\tilde{R}_k = \text{Max}^{\succ}(R_k)$ , hence there is  $S' \in \tilde{R}_k$  s.t.  $S' \succ S$ . By Definition 13,  $S' \in F$  since  $S \in F$ . Hence,  $\tilde{R}_k \cap F \neq \emptyset$  and Algorithm 2 terminates after  $k$  steps with the correct answer.

Otherwise, assume  $F$  is not reachable in  $A$ . Hence, for every  $i \geq 0$ ,  $R_i \cap F = \emptyset$ . Since  $\tilde{R}_i \subseteq R_i$  for all  $i \geq 0$ , we conclude that  $\tilde{R}_i \cap F = \emptyset$  for all  $i \geq 0$ . Hence, Algorithm 2 never returns “Reachable” in this case. It remains to show that the **repeat** loop eventually terminates. Since  $F$  is not reachable in  $A$ , there is  $k$  s.t.  $R_k = R_{k-1}$ . Hence,  $\text{Max}^{\succ}(R_k) = \text{Max}^{\succ}(R_{k-1})$ . By Lemma 14 this implies that  $\tilde{R}_k = \tilde{R}_{k-1}$ . Thus, Algorithm 2 finishes after  $k$  steps and returns “Not reachable”.  $\square$

In order to apply Algorithm 2, it remains to show how to compute a simulation relation, which should contain as many pairs of states as possible, since this raises the chances to avoid exploring some states during the breadth-first search. It is well-known that the largest simulation relation of an automaton can be computed in polynomial time wrt the size of the automaton [15]. However, this requires first computing the whole automaton, which is exactly what we want to avoid in our case. So we need to define simulations relations that can be computed *a priori*, only by considering the structure of the states (in our case, the functions  $\text{nat}$  and  $\text{rct}$ ). This is the purpose of the next section.

## 5. Idle tasks simulation relation

In this section we define a simulation relation  $\succ_{idle}$ , called the *idle tasks simulation relation* that can be computed by inspecting the values  $\text{nat}$  and  $\text{rct}$  stored in the states.

**Definition 16.** Let  $\tau$  be a set of sporadic tasks. Then, the *idle tasks preorder*  $\succ_{idle} \subseteq \text{States}(\tau) \times \text{States}(\tau)$  is s.t. for all  $S_1, S_2$ :  $S_1 \succ_{idle} S_2$  iff

1.  $\text{rct}_{S_1} = \text{rct}_{S_2}$  ;
2. for all  $\tau_i$  s.t.  $\text{rct}_{S_1}(\tau_i) = 0$ :  $\text{nat}_{S_1}(\tau_i) \leq \text{nat}_{S_2}(\tau_i)$  ;
3. for all  $\tau_i$  s.t.  $\text{rct}_{S_1}(\tau_i) > 0$ :  $\text{nat}_{S_1}(\tau_i) = \text{nat}_{S_2}(\tau_i)$ .

Notice the relation is reflexive as well as transitive, and thus indeed a preorder. It also defines a partial order on States( $\tau$ ) as it is antisymmetric. Moreover, since  $S_1 \succ_{idle} S_2$  implies that  $\text{rct}_{S_1} = \text{rct}_{S_2}$ , we also have  $\text{Active}(S_1) = \text{Active}(S_2)$ . Intuitively, a state  $S_1$  simulates a state  $S_2$  iff (i)  $S_1$  and  $S_2$  coincide on all the active tasks (i.e., the tasks  $\tau_i$  s.t.  $\text{rct}_{S_1}(\tau_i) > 0$ ), and (ii) the  $\text{nat}$  of each idle task is not larger in  $S_1$  than in  $S_2$ . Let us show that this preorder is indeed a simulation relation when we consider a *memoryless* scheduler (which is often the case in practice):

**Theorem 17.** Let  $\tau$  be a set of sporadic tasks and let Run be a memoryless (deterministic) scheduler for  $\tau$  on  $m$  processors. Then,  $\succ_{idle}$  is a simulation relation for  $A(\tau, \text{Run})$ .

*Proof.* Let  $S_1, S'_1$  and  $S_2$  be three states in States( $\tau$ ) s.t.  $(S_1, S'_1) \in E$  and  $S_2 \succ_{idle} S_1$ , and let us show that there exists  $S'_2 \in \text{States}(\tau)$  with  $(S_2, S'_2) \in E$  and  $S'_2 \succ_{idle} S'_1$ .

Since  $(S_1, S'_1) \in E$ , there exists  $\bar{S}_1$  and  $\tau' \subseteq \tau$  s.t.  $S_1 \xrightarrow{\tau'} \bar{S}_1 \xrightarrow{\text{Run}} S'_1$ , by Definition 12. Let  $\bar{S}_2$  be the (unique) state s.t.  $S_2 \xrightarrow{\tau'} \bar{S}_2$ , and let us show that  $\bar{S}_2 \succ_{idle} \bar{S}_1$ :

1. for all  $\tau_i \in \tau'$ :  $\text{rct}_{\bar{S}_1}(\tau_i) = C_i = \text{rct}_{\bar{S}_2}(\tau_i)$ . For all  $\tau_i \notin \tau'$ :  $\text{rct}_{\bar{S}_1}(\tau_i) = \text{rct}_{S_1}(\tau_i)$ ,  $\text{rct}_{\bar{S}_2}(\tau_i) = \text{rct}_{S_2}(\tau_i)$ , and, since  $S_2 \succ_{idle} S_1$ :  $\text{rct}_{S_1}(\tau_i) = \text{rct}_{S_2}(\tau_i)$ . Thus we conclude that  $\text{rct}_{\bar{S}_1} = \text{rct}_{\bar{S}_2}$ .
2. Let  $\tau_i$  be s.t.  $\text{rct}_{\bar{S}_1}(\tau_i) = 0$ . Then, we must have  $\tau_i \notin \tau'$ . In this case,  $\text{nat}_{\bar{S}_1}(\tau_i) = \text{nat}_{S_1}(\tau_i)$ ,  $\text{nat}_{\bar{S}_2}(\tau_i) = \text{nat}_{S_2}(\tau_i)$ , and, since  $S_2 \succ_{idle} S_1$ ,  $\text{nat}_{S_2}(\tau_i) \leq \text{nat}_{S_1}(\tau_i)$ . Hence,  $\text{nat}_{\bar{S}_2}(\tau_i) \leq \text{nat}_{\bar{S}_1}(\tau_i)$ . We conclude that for every  $\tau_i$  s.t.  $\text{rct}_{\bar{S}_1}(\tau_i) = 0$ :  $\text{nat}_{\bar{S}_2}(\tau_i) \leq \text{nat}_{\bar{S}_1}(\tau_i)$
3. By similar reasoning, we conclude that, for all  $\tau_i$  s.t.  $\text{rct}_{\bar{S}_1}(\tau_i) > 0$ :  $\text{nat}_{\bar{S}_1}(\tau_i) = \text{nat}_{\bar{S}_2}(\tau_i)$

Then observe that, by Definition 13,  $\bar{S}_2 \succ_{idle} \bar{S}_1$  implies that  $\text{Active}(\bar{S}_1) = \text{Active}(\bar{S}_2)$ . Let  $\tau_i$  be a task in  $\text{Active}(\bar{S}_1)$ , hence  $\text{rct}_{\bar{S}_1}(\tau_i) > 0$ . In this case, and since  $\bar{S}_2 \succ_{idle} \bar{S}_1$ , we conclude that  $\text{rct}_{\bar{S}_1}(\tau_i) = \text{rct}_{\bar{S}_2}(\tau_i)$  and  $\text{nat}_{\bar{S}_1}(\tau_i) = \text{nat}_{\bar{S}_2}(\tau_i)$ . Thus, since Run is memoryless by hypothesis,  $\text{Run}(\bar{S}_1) = \text{Run}(\bar{S}_2)$ , by Definition 7. Let  $S'_2$  be the unique state s.t.  $\bar{S}_2 \xrightarrow{\text{Run}} S'_2$ , and let us show that  $S'_2 \succ_{idle} S'_1$ :

1. Since  $\bar{S}_2 \succ_{idle} \bar{S}_1$ , we know that  $\text{rct}_{\bar{S}_1} = \text{rct}_{\bar{S}_2}$ . Let  $\tau_i$  be a task in  $\text{Run}(\bar{S}_1) = \text{Run}(\bar{S}_2)$ . By Definition 10:  $\text{rct}_{S'_1}(\tau_i) = \text{rct}_{\bar{S}_1}(\tau_i) - 1$  and  $\text{rct}_{S'_2}(\tau_i) =$

$\text{rct}_{\bar{S}_2}(\tau_i) - 1$ . Hence,  $\text{rct}_{S'_1}(\tau_i) = \text{rct}_{S'_2}(\tau_i)$ . For a task  $\tau_i \notin \text{Run}(\bar{S}_1) = \text{Run}(\bar{S}_2)$ , we have  $\text{rct}_{S'_1}(\tau_i) = \text{rct}_{\bar{S}_1}(\tau_i)$  and  $\text{rct}_{S'_2}(\tau_i) = \text{rct}_{\bar{S}_2}(\tau_i)$ , again by Definition 10. Hence,  $\text{rct}_{S'_1}(\tau_i) = \text{rct}_{S'_2}(\tau_i)$ . We conclude that  $\text{rct}_{S'_1} = \text{rct}_{S'_2}$ .

2. Let  $\tau_i$  be a task s.t.  $\text{rct}_{S'_1}(\tau_i) = 0$ . By Definition 10:  $\text{nat}_{S'_1}(\tau_i) = \max\{0, \text{nat}_{\bar{S}_1}(\tau_i) - 1\}$  and  $\text{nat}_{S'_2}(\tau_i) = \max\{0, \text{nat}_{\bar{S}_2}(\tau_i) - 1\}$ . However, since  $\bar{S}_2 \succ_{\text{idle}} \bar{S}_1$ , we know that  $\text{nat}_{\bar{S}_1}(\tau_i) \leq \text{nat}_{\bar{S}_2}(\tau_i)$ . We conclude that  $\text{nat}_{S'_1}(\tau_i) \leq \text{nat}_{S'_2}(\tau_i)$ .
3. Let  $\tau_i$  be a task s.t.  $\text{rct}_{S'_1}(\tau_i) > 0$ . By Definition 10:  $\text{nat}_{S'_1}(\tau_i) = \max\{0, \text{nat}_{\bar{S}_1}(\tau_i) - 1\}$  and  $\text{nat}_{S'_2}(\tau_i) = \max\{0, \text{nat}_{\bar{S}_2}(\tau_i) - 1\}$ . Since  $\text{rct}_{S'_1}(\tau_i) > 0$ , we have  $\text{rct}_{\bar{S}_1}(\tau_i) > 0$  too, since  $\text{rct}$  can only decrease with time elapsing. Since  $S_1 \succ_{\text{idle}} S_2$  we have also  $\text{nat}_{\bar{S}_2}(\tau_i) = \text{nat}_{\bar{S}_1}(\tau_i)$ . We conclude that  $\text{nat}_{S'_1}(\tau_i) = \text{nat}_{S'_2}(\tau_i)$ .

To conclude the proof it remains to show that, if  $S_2 \succ_{\text{idle}} S_1$  and  $S_1 \in \text{Fail}_\tau$  then  $S_2 \in \text{Fail}_\tau$  too. Let  $\tau_i$  be a task s.t.  $\text{laxity}_{S_1}(\tau_i) = \text{nat}_{S_1}(\tau_i) - (T_i - D_i) - \text{rct}_{S_1}(\tau_i) < 0$ . Since  $S_2 \succ_{\text{idle}} S_1$ :  $\text{rct}_{S_2}(\tau_i) = \text{rct}_{S_1}(\tau_i)$ , and  $\text{nat}_{S_2}(\tau_i) \leq \text{nat}_{S_1}(\tau_i)$ . Hence,  $\text{laxity}_{S_2}(\tau_i) = \text{nat}_{S_2}(\tau_i) - (T_i - D_i) - \text{rct}_{S_2}(\tau_i) \leq \text{laxity}_{S_1}(\tau_i) < 0$ , and thus,  $S_2 \in \text{Fail}_\tau$ .  $\square$

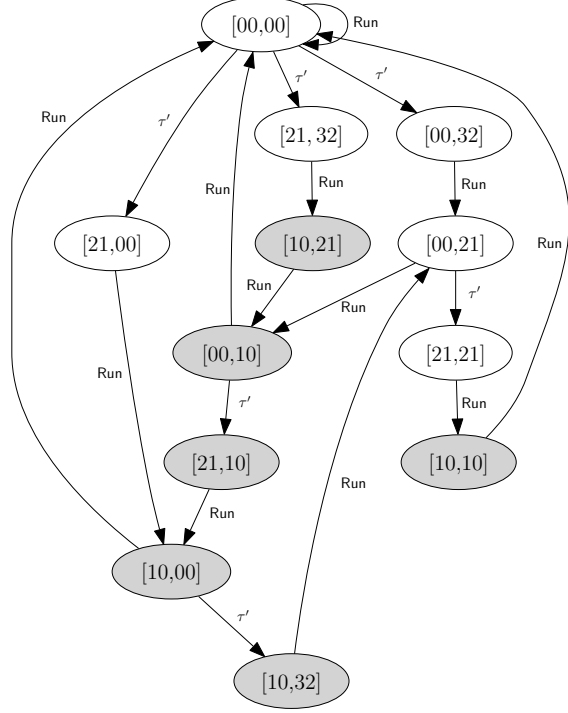
Note that Theorem 17 does *not* require the scheduler to be work-conserving. Theorem 17 tells us that any state where tasks have to wait until their next job release can be simulated by a corresponding state where they can release their job earlier, regardless of the specifics of the scheduling policy as long as it is deterministic, predictable and memoryless, which is what many popular schedulers are in practice, such as preemptive DM or EDF.

Figure 1, previously presented in Section 2, illustrates the effect of using  $\succ_{\text{idle}}$  with Algorithm 2. If a state  $S_1$  has been encountered previously and we find another state  $S_2$  such that  $S_1 \succ_{\text{idle}} S_2$ , then we can avoid exploring  $S_2$  and its successors altogether. However, note that this does not mean we will never encounter a successor of  $S_2$  as they may be encountered through other paths (or indeed, may have been encountered already).

## 6. Experimental results

We implemented both Algorithm 1 (denoted *BF*) and Algorithm 2 (denoted *ACBF* for “antichain breadth-first”) in C++ using the STL and Boost libraries 1.40.0. We ran head-to-head tests on a system equipped with a quad-core 3.2 GHz Intel Core i7 processor and 12 GB of RAM running under Ubuntu Linux 8.10 for AMD64. Our programs were compiled with Ubuntu’s distribution of GNU g++ 4.4.5 with flags for maximal optimization.

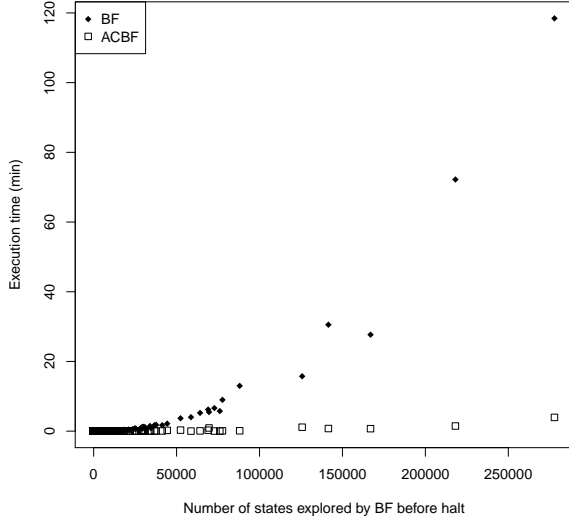
We based our experimental protocol on that used in [3]. We generated random task sets where task minimum interarrival times  $T_i$  were uniformly distributed



**Figure 1. Algorithm 2 exploits simulation relations to avoid exploring states needlessly. With  $\succ_{\text{idle}}$  on this small example, all grey states can be avoided as they are simulated by another state (e.g.  $[00, 21] \succ_{\text{idle}} [10, 21]$  and  $[00, 00] \succ_{\text{idle}} [10, 10]$ ).**

in  $\{1, 2, \dots, T_{\max}\}$ , task WCETs  $C_i$  followed an exponential distribution of mean  $0.35 T_i$  and relative deadlines were uniformly distributed in  $\{C_i, \dots, T_i\}$ . Task sets where  $n \leq m$  were dropped as well as sets where  $\sum_i C_i / T_i > m$ . Duplicate task sets were discarded as were sets which could be scaled down by an integer factor. We used EDF as scheduler and simulated  $m = 2$  for all experiments. Execution times (specifically, used CPU time) were measured using the `C clock()` primitive.

Our first experiment used  $T_{\max} = 6$  and we generated 5,000 task sets following the previous rules (of which 3,240 were EDF-schedulable). Figure 2 showcases the performance of both algorithms on these sets. The number of states explored by BF before halting gives a notion of how big the automaton was (if no failure state is reachable, the number is exactly the number of states in the automaton that are reachable from the initial state; if a failure state is reachable, BF halts before exploring the whole system). It can be seen that while ACBF and BF show similar performance for fairly small systems (roughly up to 25,000 states), ACBF outperforms BF for larger systems, and we can thus conclude that the antichains technique *scales better*. The largest system analyzed in this experiment was schedulable (and BF thus had to explore it completely),

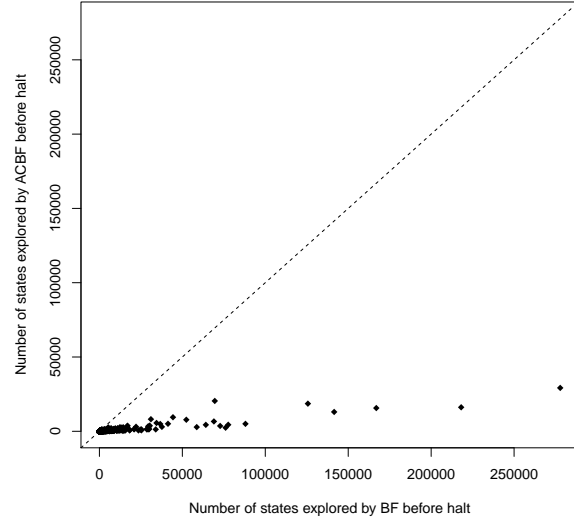


**Figure 2. States explored by BF before halt vs. execution time of BF and ACBF (5,000 task sets with  $T_{\max} = 6$ ).**

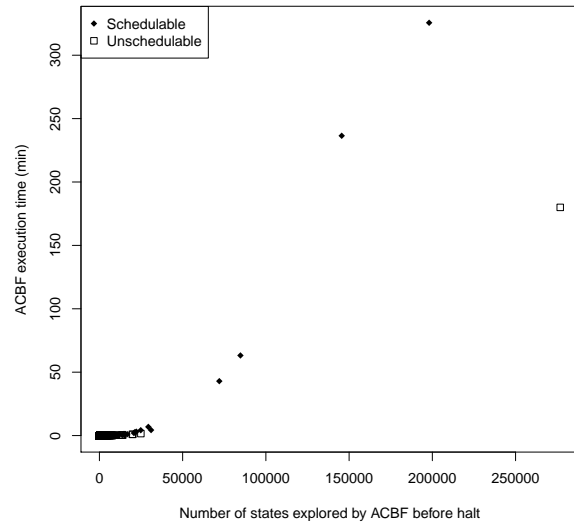
contained 277,811 states and was handled in slightly less than 2 hours with BF, whereas ACBF clocked in at 4 minutes.

Figure 3 shows, for the same experiment, a comparison between explored states by BF and ACBF. This comparison is more objective than the previous one, as it does not account for the actual efficiency of our crude implementations. As can be seen, the simulation relation allows ACBF to drop a considerable amount of states from its exploration as compared with BF: on average, 70.8% were avoided (64.0% in the case of unschedulable systems which cause an early halt, 74.5% in the case of schedulable systems). This of course largely explains the better performance of ACBF, but we must also take into account the overhead due to the more complex algorithm. In fact, we found that in some cases, ACBF would yield worse performance than BF. However, to the best of our knowledge, this only seems to occur in cases where BF took relatively little time to execute (less than five seconds) and is thus of no concern in practice.

Our second experiment used 5,000 randomly generated task sets using  $T_{\max} = 8$  (of which 3,175 were schedulable) and was intended to give a rough idea of the limits of our current ACBF implementation. Figure 4 plots the number of states explored by ACBF before halting versus its execution time. We can first notice the plot looks remarkably similar to BF in Figure 2, which seems to confirm the exponential complexity of ACBF which we predicted. The largest schedulable system considered necessitated exploring 198,072 states and required roughly 5.5 hours. As a spot-check, we ran BF on a schedulable sys-



**Figure 3. States explored by BF before halt vs. states explored by ACBF before halt (5,000 task sets with  $T_{\max} = 6$ ).**



**Figure 4. States explored by ACBF before halt vs. ACBF execution time (5,000 task sets with  $T_{\max} = 8$ ).**



tem where ACBF halted after exploring 14,754 states in 78 seconds; BF converged after just over 6 hours, exploring 434,086 states.

Our experimental results thus yield several interesting observations. The number of states explored by ACBF using the idle tasks simulation relation is significantly less on average than BF. This gives an objective metric to quantify the computational performance gains made by ACBF wrt BF. In practice using our implementation, ACBF outperforms BF for any reasonably-sized automaton, but we have seen that while our current implementation of ACBF defeats BF, it gets slow itself for slightly more complicated task sets. However, we expect smarter implementations and more powerful simulation relations to push ACBF much further.

## 7. Conclusions and future work

We have successfully adapted a novel algorithmic technique developed by the formal verification community, known as antichain algorithms [9, 11], to greatly improve the performance of an existing exact schedulability test for sporadic hard real-time tasks on identical multiprocessor platforms [3]. To achieve this, we developed and proved the correctness of a simulation relation on a formal model of the scheduling problem. While our algorithm has the same worst-case performance as a naive approach, we have shown experimentally that our preliminary implementation can still outperform the latter in practice.

The model introduced in Section 3 yields the added contribution of bringing a fully formalized description of the scheduling problem we considered. This allowed us to formally define various scheduling concepts such as memorylessness, work-conserving scheduling and various scheduling policies. These definitions are univocal and not open to interpretation, which we believe is an important consequence. We also clearly define what an execution of the system is, as any execution is a possibly infinite path in the automaton, and all possible executions are accounted for.

We expect to extend these results to the general Baker-Cirinei automaton which allows for arbitrary deadlines in due time. We chose to focus on constrained deadlines in this paper mainly because it simplified the automaton and made our proofs simpler, but we expect the extension to arbitrary deadlines to be fairly straightforward. We also only focused on developing *forward simulations*, but there also exist antichain algorithms that use *backward simulations* [11]. It would be interesting to research such relations and compare the efficiency of those algorithms with that presented in this paper.

The task model introduced in Section 2 can be further extended to enable study of more complex problems, such as job-level parallelism and semi-partitioned scheduling. The model introduced in Section 3 can also be extended to support broader classes of schedulers. This was briefly touched on in [3]. For example, storing the previous

scheduling choice in each state would allow modelling of non-preemptive schedulers.

It has not yet been attempted to properly optimize our antichain algorithm by harnessing adequate data structures; our objective in this work was primarily to get a preliminary “proof-of-concept” comparison of the performance of the naive and antichain algorithms. Adequate implementation of structures such as *binary decision diagrams* [7] and *covering sharing trees* [10] should allow pushing the limits of the antichain algorithm’s performance.

Antichain algorithms should terminate quicker by using coarser simulation preorders. Researching other simulation preorders on our model, particularly preorders that are a function of the chosen scheduling policy, is also key to improving performance. Determining the complexity class of sporadic task set feasibility on identical multiprocessor platforms is also of interest, as it may tell us whether other approaches could be used to solve the problem.

## A. Proof of Lemma 14

In order to establish the lemma, we first show that, for any set  $B$  of states, the following holds:

$$\text{Lemma 18. } \text{Max}^{\succ} \left( \text{Succ} \left( \text{Max}^{\succ} (B) \right) \right) = \text{Max}^{\succ} (\text{Succ} (B)).$$

*Proof.* We first show that  $\text{Max}^{\succ} \left( \text{Succ} \left( \text{Max}^{\succ} (B) \right) \right) \subseteq \text{Max}^{\succ} (\text{Succ} (B))$ . By def of  $\text{Max}^{\succ} (B)$ , we know that  $\text{Max}^{\succ} (B) \subseteq A$ . Moreover,  $\text{Succ}$  and  $\text{Max}^{\succ}$  are monotonic wrt set inclusion. Hence:

$$\begin{aligned} & \text{Max}^{\succ} (B) \subseteq B \\ \Rightarrow & \text{Succ} \left( \text{Max}^{\succ} (B) \right) \subseteq \text{Succ} (B) \\ \Rightarrow & \text{Max}^{\succ} \left( \text{Succ} \left( \text{Max}^{\succ} (B) \right) \right) \subseteq \text{Max}^{\succ} (\text{Succ} (B)) \end{aligned}$$

Then, we show that  $\text{Max}^{\succ} \left( \text{Succ} \left( \text{Max}^{\succ} (B) \right) \right) \supseteq \text{Max}^{\succ} (\text{Succ} (B))$ . Let  $S_2$  be a state in  $\text{Max}^{\succ} (\text{Succ} (B))$ . Let  $S_1 \in B$  be a state s.t.  $(S_1, S_2) \in E$ . Since,  $S_2 \in \text{Succ} (B)$ ,  $S_1$  always exists. Since  $S_1 \in B$ , there exists  $S_3 \in \text{Max}^{\succ} (B)$  s.t.  $S_3 \succ S_1$ . By Definition 13, there is  $S_4 \in \text{Succ} \left( \text{Max}^{\succ} (B) \right)$  s.t.  $S_4 \succ S_2$ . To conclude, let us show *per absurdum* that  $S_4$  is maximal in  $\text{Succ} \left( \text{Max}^{\succ} (B) \right)$ . Assume there exists  $S_5 \in \text{Succ} \left( \text{Max}^{\succ} (B) \right)$  s.t.  $S_5 \succ S_4$ . Since  $\text{Max}^{\succ} (B) \subseteq A$ ,  $S_5$  is in  $\text{Succ} (B)$  too. Moreover, since  $S_4 \succ S_2$  and  $S_5 \succ S_4$ , we conclude that  $S_5 \succ S_2$ . Thus, there is, in  $\text{Succ} (B)$  and element  $S_5 \succ S_2$ . This contradicts our hypothesis that  $S_2 \in \text{Max}^{\succ} (\text{Succ} (B))$ .  $\square$

Then, we are ready to show that:

Induction hypotesis we assume that  $\tilde{R}_{k-1} = \text{Max}^{\succ} (R_k)$ . Then:

$$\begin{aligned}
& \tilde{R}_k \\
= & \text{Max}^{\succ} \left( \tilde{R}_{k-1} \cup \text{Succ} \left( \tilde{R}_{k-1} \right) \right) && \text{By def.} \\
= & \text{Max}^{\succ} \left( \text{Max}^{\succ} \left( \tilde{R}_{k-1} \right) \cup \text{Max}^{\succ} \left( \text{Succ} \left( \tilde{R}_{k-1} \right) \right) \right) && \text{by (1)} \\
= & \text{Max}^{\succ} \left( \text{Max}^{\succ} \left( \text{Max}^{\succ} (R_{k-1}) \right) \right) \cup \text{Max}^{\succ} \left( \text{Succ} \left( \text{Max}^{\succ} (R_{k-1}) \right) \right) && \text{By I.H.} \\
= & \text{Max}^{\succ} \left( \text{Max}^{\succ} (R_{k-1}) \cup \text{Max}^{\succ} (\text{Succ} (R_{k-1})) \right) && \text{By Lemma 18} \\
= & \text{Max}^{\succ} (R_{k-1} \cup \text{Succ} (R_{k-1})) && \text{By (1)} \\
= & \text{Max}^{\succ} (R_k) && \text{By def.}
\end{aligned}$$

**Figure 5. Inductive case for Lemma 19.**

**Lemma 19.** Let  $A$  be an automaton and let  $\succ$  be a simulation relation for  $A$ . Let  $R_0, R_1, \dots$  and  $\tilde{R}_0, \tilde{R}_1, \dots$  denote respectively the sequence of sets computed by Algorithm 1 and Algorithm 2 on  $A$ . Then, for all  $i \geq 0$ :  $\tilde{R}_i = \text{Max}^{\succ} (R_i)$ .

*Proof.* The proof is by induction on  $i$ . We first observe that for any pair of sets  $B$  and  $C$ , the following holds:

$$\begin{aligned}
& \text{Max}^{\succ} (B \cup C) \\
= & \text{Max}^{\succ} \left( \text{Max}^{\succ} (B) \cup \text{Max}^{\succ} (C) \right) \quad (1)
\end{aligned}$$

**Base case**  $i = 0$  Clearly,  $\text{Max}^{\succ} (R_0) = R_0$  since  $R_0$  is a singleton. By definition  $\tilde{R}_0 = R_0$ .

**Inductive case**  $i = k$  See Figure 5. □

**Acknowledgment.** We thank Phan Hiep Tuan for identifying mistakes in Definition 13 and Theorem 17.

## References

- [1] Y. Abdeddaïm and O. Maler. Preemptive job-shop scheduling using stopwatch automata. In *AIPS-02 Workshop on Planning via Model-Checking, Toulouse, France*, pages 7–13, 2002.
- [2] T. P. Baker and S. K. Baruah. Schedulability analysis of multiprocessor sporadic task systems. In I. Lee, J. Y.-T. Leung, and S. Son, editors, *Handbook of Real-Time and Embedded Systems*. Chapman & Hall/CRC Press, 2007.
- [3] T. P. Baker and M. Cirinei. Brute-force determination of multiprocessor schedulability for sets of sporadic hard-deadline tasks. In Tovar et al. [17], pages 62–75.
- [4] S. K. Baruah and N. Fisher. Global deadline-monotonic scheduling of arbitrary-deadline sporadic task systems. In Tovar et al. [17], pages 204–216.
- [5] M. Bertogna and S. K. Baruah. Tests for global EDF schedulability analysis. *Journal of Systems Architecture - Embedded Systems Design*, 57(5):487–497, 2011.
- [6] V. Bonifaci and A. Marchetti-Spaccamela. Feasibility analysis of sporadic real-time multiprocessor task systems. In M. de Berg and U. Meyer, editors, *ESA (2)*, volume 6347 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2010.
- [7] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, 1992.
- [8] F. Cassez. Timed games for computing WCET for pipelined processors with caches. In *11th Int. Conf. on Application of Concurrency to System Design (ACSD’2011)*. IEEE Computer Society, June 2011. Forthcoming.
- [9] M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In T. Ball and R. B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 17–30. Springer, 2006.
- [10] G. Delzanno, J.-F. Raskin, and L. Van Begin. Covering sharing trees: a compact data structure for parameterized verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):268–297, 2004.
- [11] L. Doyen and J.-F. Raskin. Antichain algorithms for finite automata. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *Lecture Notes in Computer Science*, pages 2–22. Springer, 2010.
- [12] E. Fersman, P. Krcal, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Inf. Comput.*, 205(8):1149–1172, 2007.
- [13] E. Filot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 263–277. Springer, 2009.
- [14] J. Goossens, S. Funk, and S. Baruah. EDF scheduling on multiprocessor platforms: some (perhaps) counterintuitive observations. In *Proceedings of the eighth International Conference on Real-time Computing Systems and Applications (RTCSA)*, pages 321–330, Tokyo Japan, March 2002.
- [15] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462, 1995.
- [16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.
- [17] E. Tovar, P. Tsigas, and H. Fouchal, editors. *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17–20, 2007. Proceedings*, volume 4878 of *Lecture Notes in Computer Science*. Springer, 2007.